attacklab 实验报告

Tan Hengkai

2021 年 10 月 23 日

1 实验目的

1. 学习几种攻击程序的方法，包括注入代码攻击，和 Return-Oriented Programming 攻击

2. 深入学习 gdb 和 objdump 的使用

3. 深入理解栈和寄存器等汇编程序设计知识，以及 x86-64 指令及其机器码

4. 理解程序安全隐患与汇编的联系

2 实验原理

2.1 指令、程序计数器与栈

程序的每一条指令都有对应的内存地址表示，程序计数器中保存下一条指令在内存中的地址。栈中的每个位置也可以用一个十六进制数地址来表示。

2.2 return

return 时将%rsp 当前指向的 return address 放入程序计数器，作为下次执行的命令地址；然后将%rsp 向上挪 8 bytes。

所以我只需修改 return address，即可让程序在%rsp 到达这个被篡改的 return address 之后跳转到目标函数。同样，可以在栈中指定位置写入一段代码，记录代码所在地址，修改 return address 也可让程序执行指定的代码。

2.3 gadget

一个 gadget 形如指令 +c3，表示执行一个指令之后 return。这个指令的机器代码存在于程序的机器代码的某个子串中。

从某一个 return address 开始写入 gadgets，当%rsp 移动到这个 return address 并 return 时，程序会执行 gadget 对应的代码，执行 gadget 里面的 return 之后，程序紧接着又会执行此 gadget 上方的 gadget，如此下去即可从下往上地执行一连串的隐藏代码。

3 实验过程

先用 objdump 进行反汇编，可以看到 ctarget 和 rtarget 的机器码及对应的汇编代码，放于 ctarget.d 和 rtarget.d 文件中。

思路：调用 getbuf 函数；往 buf 中输入攻击字符串；getbuf 函数返回时，进入指定 touch 函数的指定分支。

3.1 Code Injection Attacks

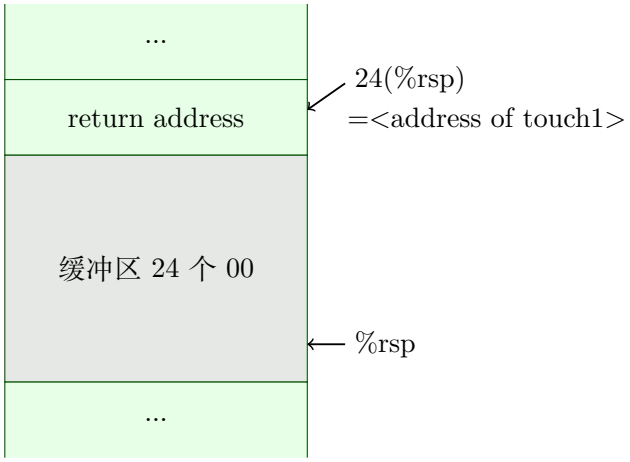
往 buf 中直接注入一段可以执行的机器代码，并通过修改 return address 来使程序执行注入的攻击代码。

3.1.1 Phase 1

通过观察 ctarget.d 可以知道，在 getbuf 函数中，rsp 增加了 24 ，在这块开辟的栈空间缓冲区上方，便是 return address，正常执行的 getbuf 函数会返回到 call getbuf 的下一条语句，因此 return address 是 call getbuf 的下一条语句地址。

那么只需要用 24 个 00（因为是十六进制，表示 24 bytes）填充这块缓冲区，紧接着放置 touch1 的函数地址作为新的 return address 即可。touch1 的函数地址可以在 ctarget.d 中找到。

读入字符串时：



3.1.2 Phase 2

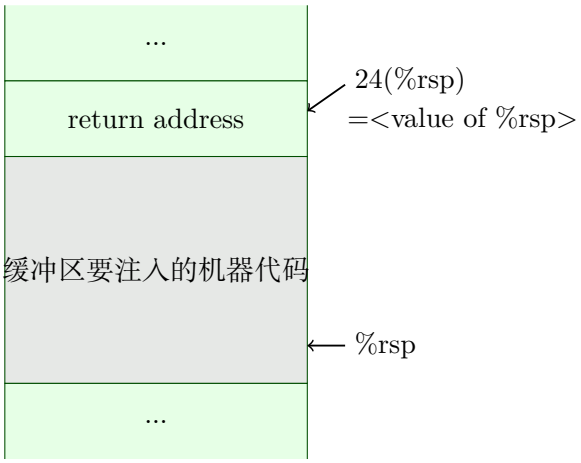
touch2 的参数 val 必须赋值为 cookie 才能进入正确的分支，那么需要对%rdi 进行赋值。

在缓冲区注入代码。在 gdb 中使用 info registers 即可查看 rsp 的值，可以看到缓冲区开始和结束的位置。将缓冲区上方的 return address 覆盖为缓冲区开始的位置（地址小的一端），这样 getbuf 返回时，程序就会执行我们注入在缓冲区的代码。为了最后进入 touch2 函数，注入代码中使用 push <address of touch2> 和 ret，即可在执行 ret 之后进入 touch2 函数。

```
movq $0x3bf83a01,%rdi  # cookie
push $0x40190d   # address of touch2
retq
```

可以将要注入的代码汇编代码使用 gcc 转换成.o 文件，然后使用 objdump 即可得到注入代码的机器码表示。将此机器码作为输入。

读入字符串时：

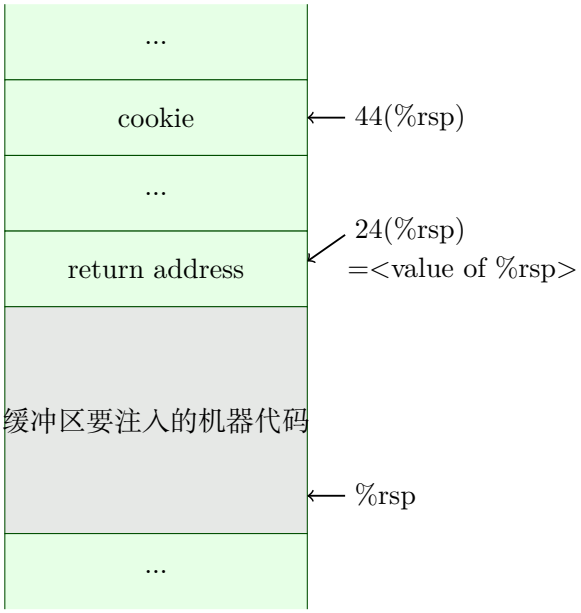


3.1.3 Phase 3

这里 touch3 的参数是 char* 类型的，意味着我们需要将 cookie 字符串存在内存中，然后将 rdi 赋值为 cookie 字符串的起始位置。将 cookie 字符串转换为二进制表示（使用 ascii -x 可以查看每种字符的十六进制代码），写在缓冲区上方（也是 getbuf 的 return address 上方），并记下 cookie 的起始位置。其他过程与 Phase 2 同理。

```
1 movq $0x55652ea8,%rdi  # starting address of cookie
2 push $0x401a24  # address of touch3
3 retq
```

读入字符串时：



3.2 Return-Oriented Programming

利用程序中存在的机器码，选取一些子串拼凑成攻击代码，作为 gadgets，在 getbuf 函数的 return address 中向上填充 gadgets 即可执行一个个 gadget。

3.2.1 Phase 4

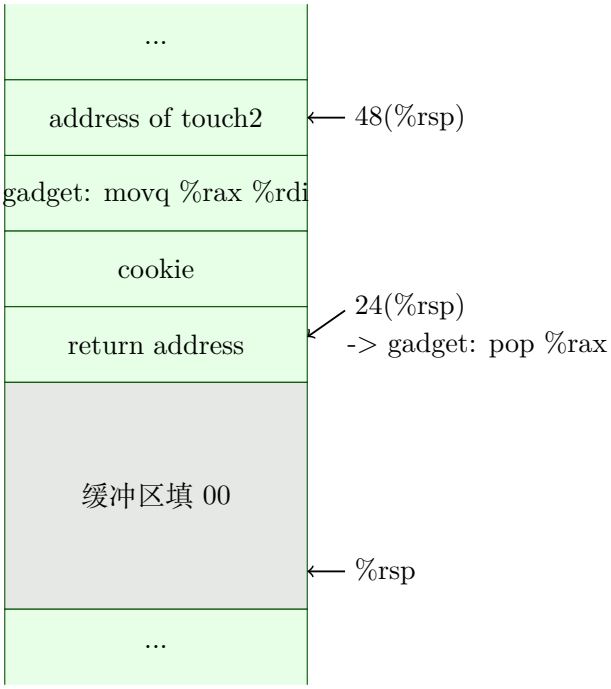
由 Phase 2 的代码：

```
1  movq <cookie> %rdi
2  push <address of touch2>
3  retq
```

改进为 gadgets 的形式（同时保证每条指令都存在于 rtarget.d 中），cookie 和 address 只需要直接作为输入即可：

```
1  pop %rax  retq
2  <cookie>
3  movq %rax, %rdi  retq
4  <address of touch2>
```

这样就可以将%rdi 赋值为 cookie，同时在最后一个 gadget 执行完毕之后，让程序运行 touch2 函数。读入字符串后：

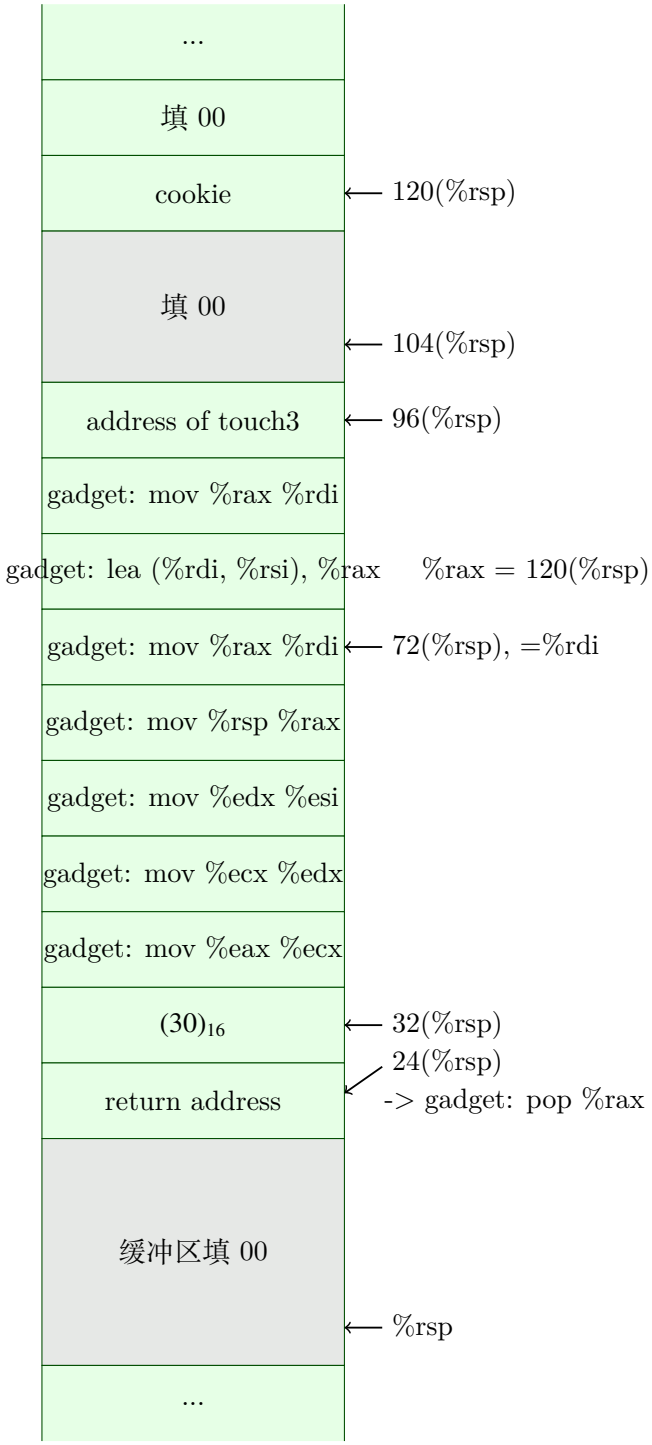


3.2.2 Phase 5

将 cookie 转化成 ascii 码保存在高位某位置，现在的任务就是将%rdi 指向 cookie 的起始位置，但是%rsp 指针不能经过 cookie 存放的位置。

观察到 rtarget.d 中存在 lea (%rdi, %rsi), %rax 指令，那么可以将%rsi 通过 pop 赋值为一个偏移量（这里设置的偏移量为 $(30)_{16} = 48$），然后%rdi 通过 mov 赋值为一个与栈顶指针%rsp 关联的量，两者相加即可得到%rax 为一个高位的值作为 cookie 的起始位置，再将%rax 赋值给%rdi 即可。赋值操作往往要拆成一连串的 mov 指令来进行，指令见下图 stack。

读入字符串后：



4 遇到的困难 & 心得 & 技巧与经验

一开始遇到的困难是，没有想到每条指令都有对应的内存地址表示。后来不断思索、回顾课上内容，想起我注入一段代码在缓冲区中，然后记下缓冲区的地址覆盖 return address，即可通过 return 的操作让程序执行缓冲区地址的代码。这使我理解到上一遍课学到的东西很难深刻理解，还要通过复习和实践来提升理解的深度。经过这个作业之后，我更加深刻地理解了栈和寄存器等汇编程序设计知识。

第二次遇到的困难是在 Phase 3 中，调试了很久总是 segmentation fault 或者 mismatch。之后突然

想起把之前记录的 cookie, address 等信息检查一遍，马上发现了 touch3 函数地址记录错误的问题。调试程序的时候也是如此，先肉眼查一遍基本的错误，往往会省下很多时间和精力。